

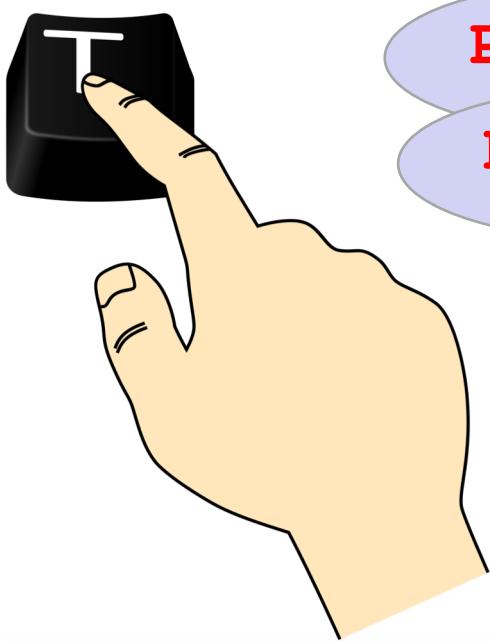
**Calcolatori Elettronici T  
Ingegneria Informatica**

# **04 Interruzioni**



## Gestione eventi con una CPU: *polling*

- In un sistema a microprocessore è di fondamentale importanza poter gestire eventi che si verificano all'esterno (ma non solo) della CPU
- Per esempio, determinare se è stato premuto un tasto sulla tastiera, se il mouse è stato spostato, etc
- Una strategia, poco efficiente, per raggiungere questo scopo consiste nel controllare periodicamente se tali eventi si sono verificati (**polling**)
- Questo può essere fatto interrogando di continuo la periferica che si desidera gestire
- Ovviamente, con questa strategia, la CPU spende molti cicli macchina per la verifica (o le verifiche)
- Una strategia molto più efficiente, basata su una strategia “push”, consiste nell’uso di **interrupt**



Premuto?

Premuto?

Premuto?

Premuto?

```
main()
{
    bool tasto_premuto=false;

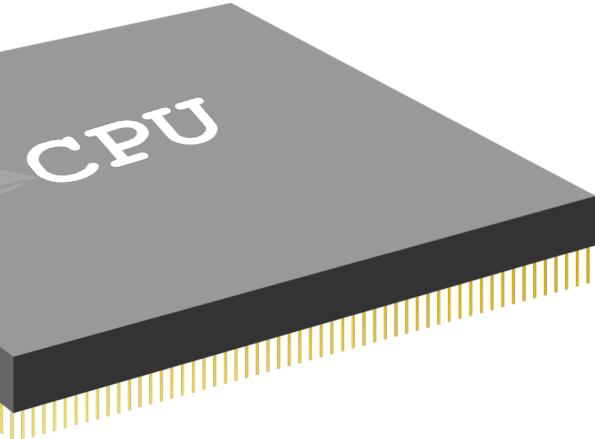
    while(1)
    {
        if (tasto_premuto==true)
            gestisci_evento();

        . . .
    }

    void gestisci_evento()
    {
        . . .
        return;
    }
}
```

La CPU spende molto tempo nel controllare (*polling*) se l'evento si è verificato. Questa strategia rallenta l'esecuzione del main

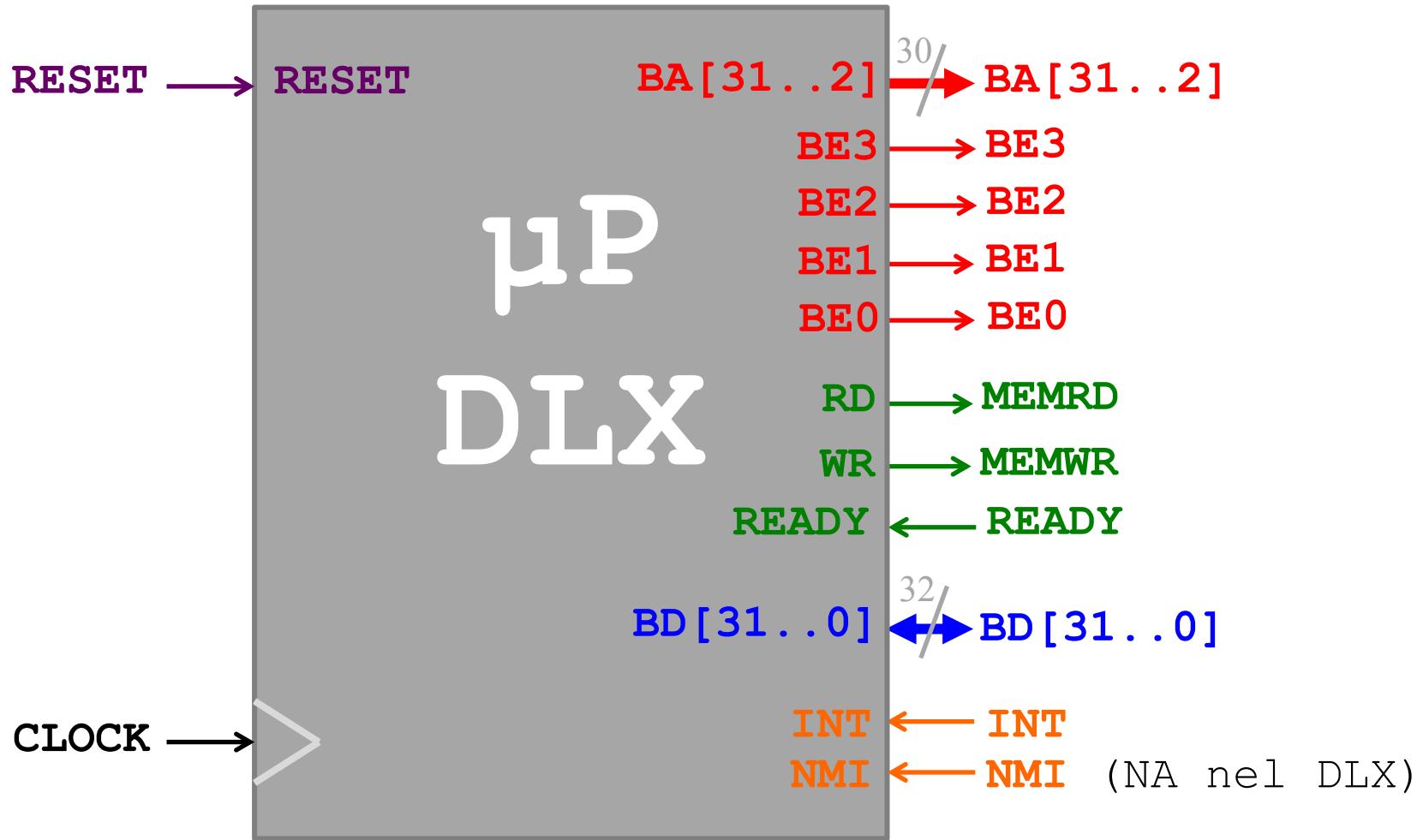
Poco efficiente....



## Gestione eventi con CPU: *interrupt*

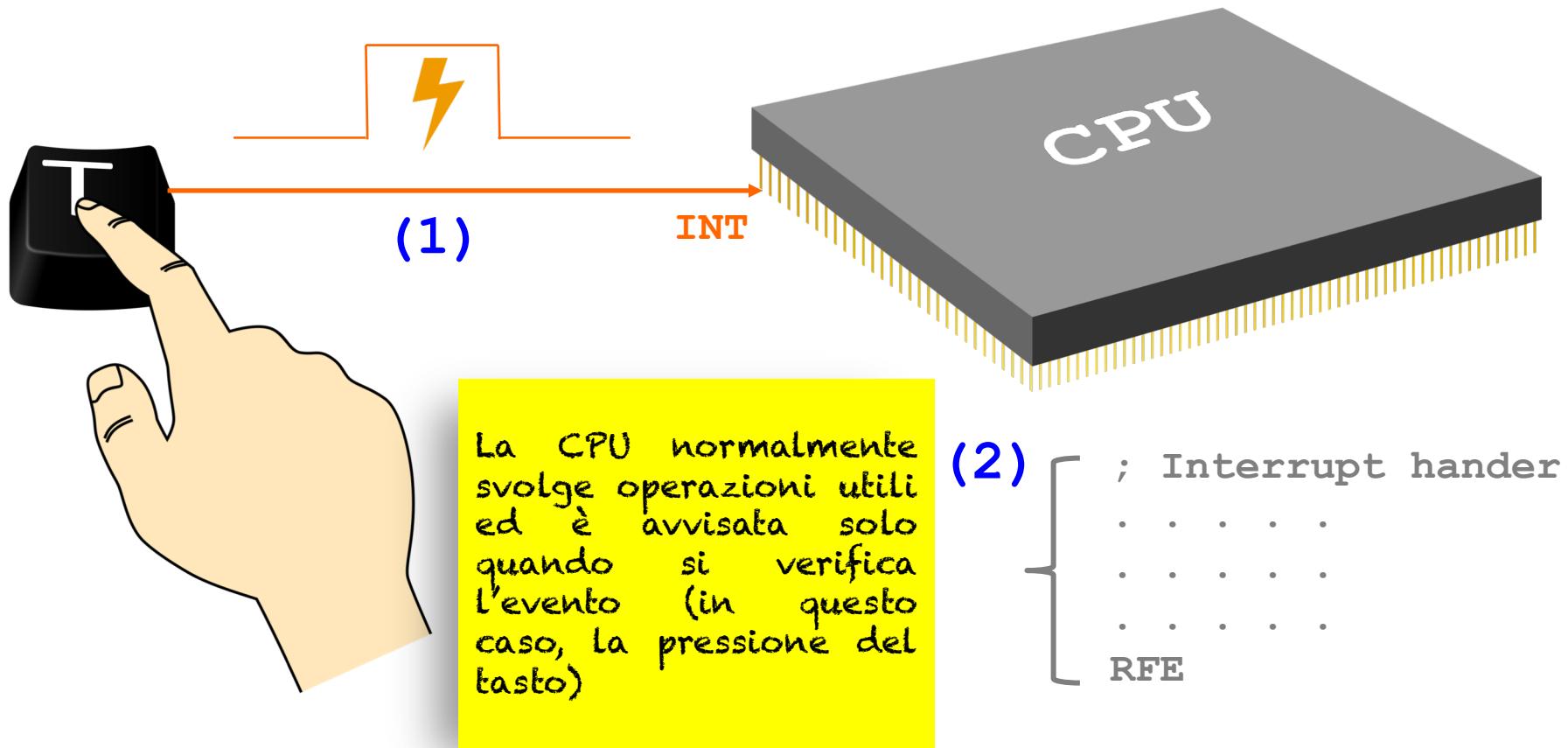
- Un *interrupt* è un evento che interrompe la CPU durante il regolare flusso di esecuzione del codice
- L' interrupt segnala che si è verificato un evento che merita **immediata\* attenzione** da parte della CPU
- Se la CPU è abilitata\* a ricevere tale segnalazione, esegue automaticamente una porzione di codice denominata *interrupt handler* al fine di gestire l'evento
- Gli eventi possono essere relativi a **fattori esterni** (e.g., premuto un tasto) o **interni** (e.g., è stata eseguita una divisione per zero, overflow, etc)
- Quando dipendono da fattori interni si parla di **eccezioni** (*exceptions*)
- Inoltre, è possibile invocare l'handler mediante opportune istruzioni (e.g., per invocare *system call*)

# Gestione interruzioni nel DLX



In ogni processore, è presente almeno un segnale denominato **INT** per gestire le interruzioni. In molti casi, ma non nel DLX, è presente anche un ulteriore segnale denominato **NMI** per gestire interruzioni che non possono essere ignorate.

- Nel caso di *interrupt* generato dall'esterno la situazione è questa:



- La pressione del tasto innesca\* l'esecuzione del codice dell' interrupt handler (2)

- Nel caso di *interrupt* generato dall'esterno la situazione, dal punto di vista software, è questa:

```
main()
{
    Istruzione 1;
    Istruzione 2;
    Istruzione 3;
    Istruzione 4;          (i) ⚡
    Istruzione 5;          (ii)
    Istruzione 6;
    Istruzione 7;
    Istruzione 8;
}
```

L'istruzione 4 è portata a termine prima di eseguire l' interrupt handler

; Interrupt handler  
ADD R1,R0,R0  
. . . . .  
. . . . .  
RFE

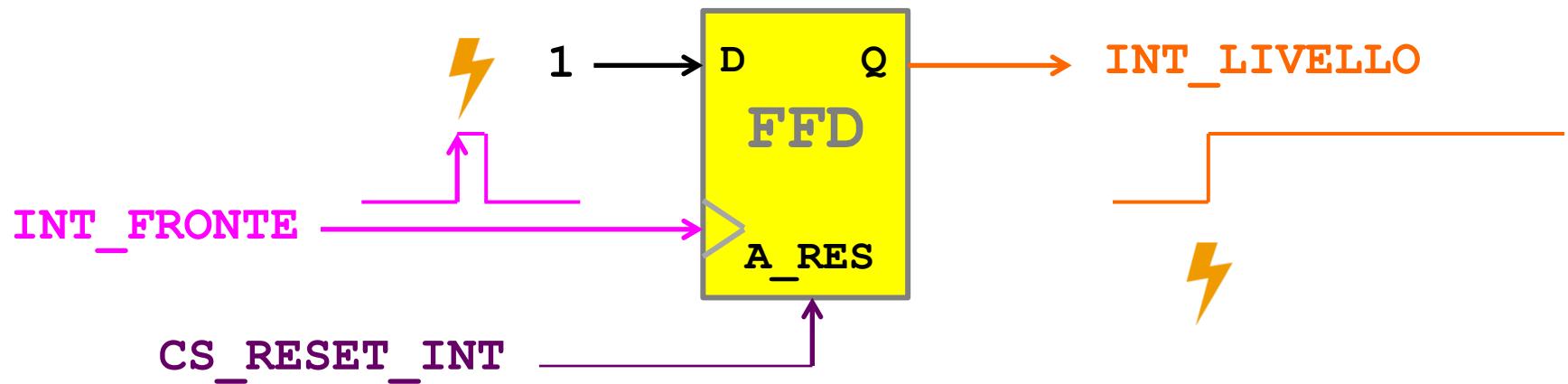
- L'*interrupt* può verificarsi in qualsiasi momento (i.e., durante l'esecuzione di qualsiasi istruzione) e non è sincronizzato con il clock
- Assumeremo sempre che, l'esecuzione dell'*istruzione* durante la quale si verifica l'*interrupt* sia sempre portata a termine prima di eseguire l'*handler*

## Segnale di interrupt: fronte o livello

- Esistono CPU sensibili al livello del segnale di interrupt, altre al fronte di salita e altre a entrambe le cose
- Nel caso del DLX assumeremo che la CPU sia sensibile al livello del segnale (1 se l'interrupt è attivo e 0 in caso contrario)
- Nel caso dei dispositivi che generano interrupt, assumeremo che esso rimanga a 1 fintantoché la causa che lo ha generato non sia stata gestita dalla CPU
- Pertanto, se una periferica ha un interrupt a livello asserito, rimane tale fintantoché l'interrupt non è gestito dalla CPU (non necessariamente subito\*)
- In alcuni casi, nell'handler può essere necessario eseguire delle operazioni software per poter portare al livello logico 0 il segnale di interrupt proveniente dall'esterno dopo aver gestito l'evento

## Trasformazione da fronte a livello

- Come fare se il dispositivo che genera l'interrupt assume che la CPU sia sensibile ai fronti mentre la CPU è sensibile solo al livello del segnale?
- E' necessario eseguire una trasformazione da fronte a livello del segnale **INT\_FRONT**
- In un caso come questo, il livello logico del segnale **INT\_LIVELLO** deve essere portato a zero da un opportuno comando software (CPU) che asserisce il segnale **CS\_RESET\_INT**

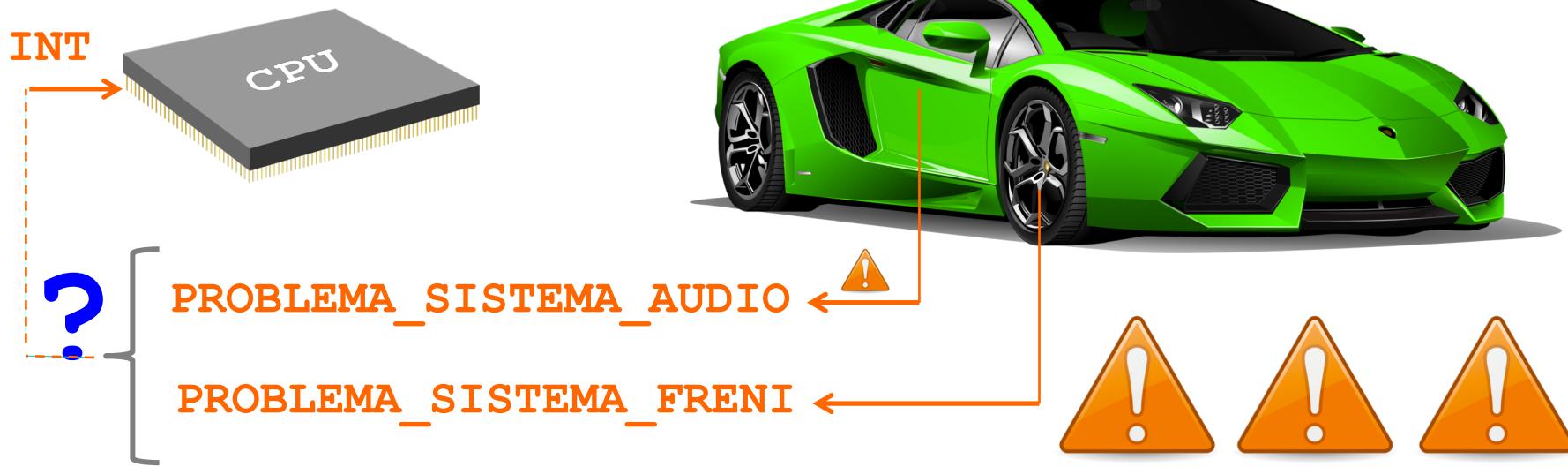


## Gestione di interruzioni multiple

- C'è però un problema: escludendo NMI (discusso dopo) il DLX ha un solo segnale di interrupt denominato INT. Come facciamo a gestire, come tipicamente accade, multiple sorgenti di interrupt?
- Si convogliano (e.g., mediante un OR o altre funzioni in base alle specifiche esigenze) tutti gli interrupt verso l'unico segnale INT presente nel DLX
- Rimane un altro problema: come determinare quale/quali interrupt sono asseriti in un determinato istante?
- A tal proposito è (tipicamente) necessario poter determinare lo stato delle richieste di interrupt mediante opportune istruzioni software
- Vedremo che esistono anche delle reti, denominate PIC, che possono agevolare questo compito alla CPU

# Interruzioni multiple e priorità

- In un sistema nel quale è presente più di una sorgente di interruzione è fondamentale poter associare un **livello di priorità** a ciascuna interruzione
- Sarebbe auspicabile poter interrompere l'**interrupt handler** in esecuzione se giunge una richiesta di interruzione più prioritaria (*annidamento*)
- Esempio:



## Interrupt nel DLX

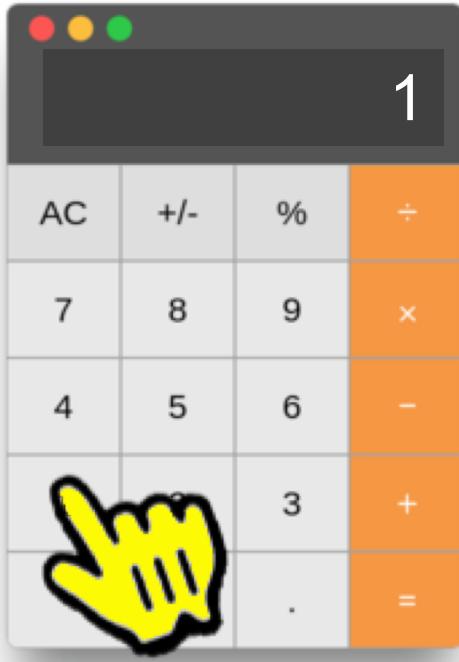
- Assumeremo che il DLX sia sensibile al **livello del segnale** di interrupt **INT** e non al suo fronte
- L'indirizzo di ritorno (PC+4) è salvato in **IAR**
- In seguito all'arrivo di un interrupt, **l'istruzione in corso** è **completata** ed è eseguito il codice all'indirizzo **00000000h**
- Il ritorno dall'interrupt handler (PC  $\leftarrow$  IAR) avviene mediante l'istruzione **RFE** (*Return From Exception*)
- In genere, ma non nel DLX base, gli interrupt possono essere abilitati o disabilitati mediante istruzioni
- Nell'ISA DLX, è gestito un solo indirizzo di ritorno. Pertanto, il DLX disabilita le interruzioni mentre esegue l'handler e le riabilita automaticamente ritornando dall'handler (RFE). In caso contrario, nel DLX, servirebbe uno **stack** software

- Con *annidamento (nesting)* delle *interruzioni* si intende la possibilità di poter avviare un interrupt handler durante l'esecuzione di un altro handler
- Questa caratteristica è standard nella maggior parte delle CPU in commercio ma **non è prevista dal DLX base**
- Per poter *annidare* gli interrupt sarebbe necessario uno *stack software* (utilizzando l'istruzione MOVS2I) e avere la possibilità di ri-abilitare gli interrupt nell'handler mediante opportune istruzioni (ENI) non prevista dall'ISA base
- In caso multiple sorgenti di interruzione, nasce il problema di come associare una scala di priorità alle interruzioni
- A tal fine esistono varie politiche: **priorità fissa**, **variabile**, etc. Ovviamente la priorità è cruciale non solo quando è possibile annidare gli interrupt

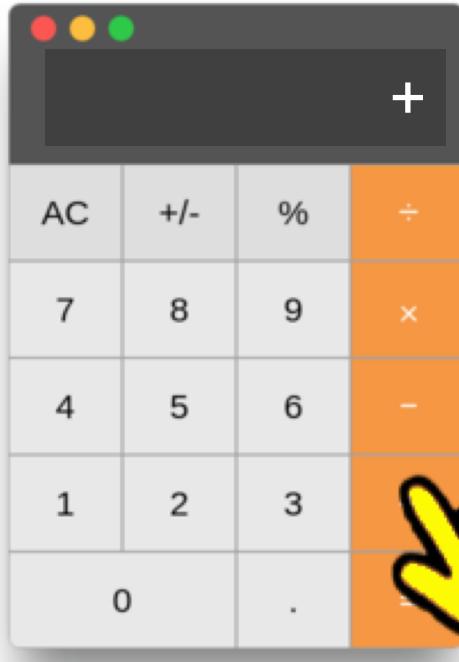
## Interrupt handler e consistenza dei dati

- Le richieste di interrupt possono verificarsi in qualsiasi momento
- E' però necessario mantenere la consistenza dei dati in modo che il codice in esecuzione non sia modificato dall'arrivo o meno di interrupt e di conseguenza dall'esecuzione o meno degli interrupt handler
- Per questa ragione è necessario fare in modo che l' interrupt handler (i.e., il driver del dispositivo) non interferisca con il codice del programma (main) in esecuzione
- Come fare? Salvando e ripristinando i registri modificati dall' interrupt handler all'interno dello stesso codice (handler)
- Nella pagina seguente è mostrato l' effetto di un pessimo interrupt handler che non preserva i registri

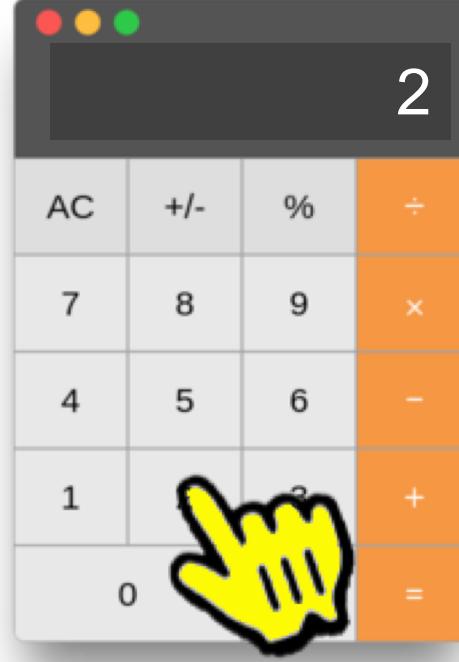
a)



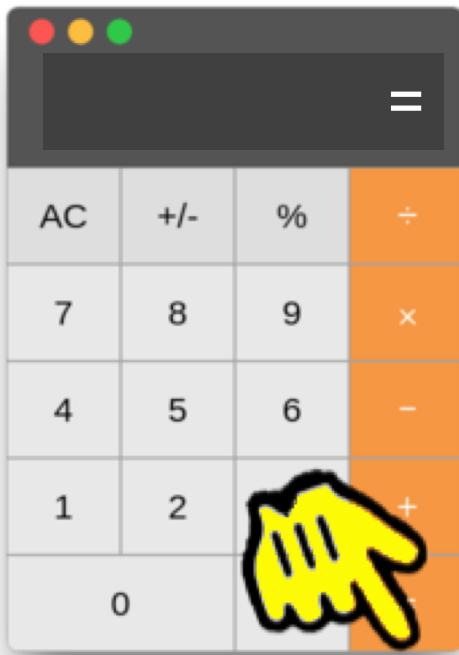
b)



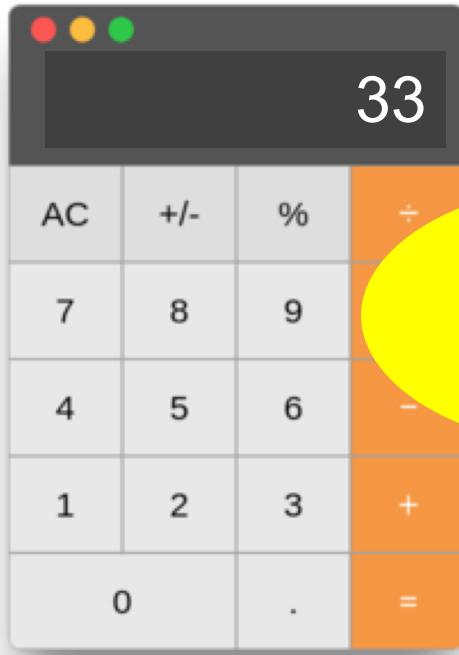
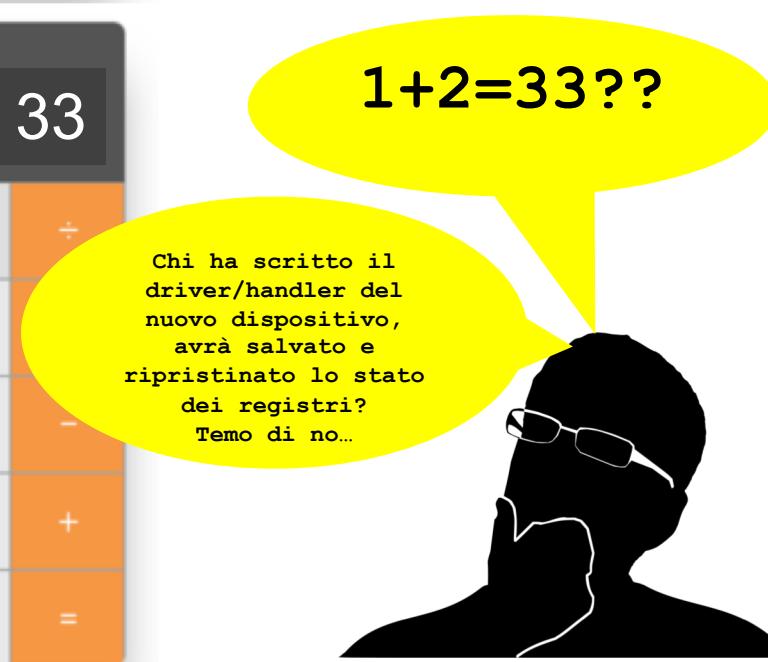
c)



d)

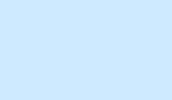
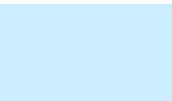
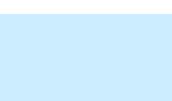


e)


 $1+2=33??$ 


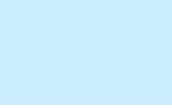
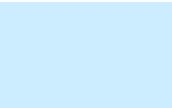
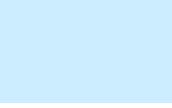
## Interrupt handler con singola interruzione

Nel caso di una **singola sorgente di interruzione**, il codice di un tipico interrupt handler potrebbe avere la struttura seguente:

|            |   |  |
|------------|---|--|
| 00000000h  | <br><br> | ; Istruzioni che salvano i registri<br>; modificati dalle istruzioni seguenti<br><br>; codice di risposta alla richiesta<br>; di interruzione<br><br>; istruzioni di ripristino dei registri<br>; modificati in precedenza |
| XXXXXXXXXh | RFE   | ; ritorno dall'interrupt (PC ← IAR)  |

# Interrupt handler con multiple interruzione

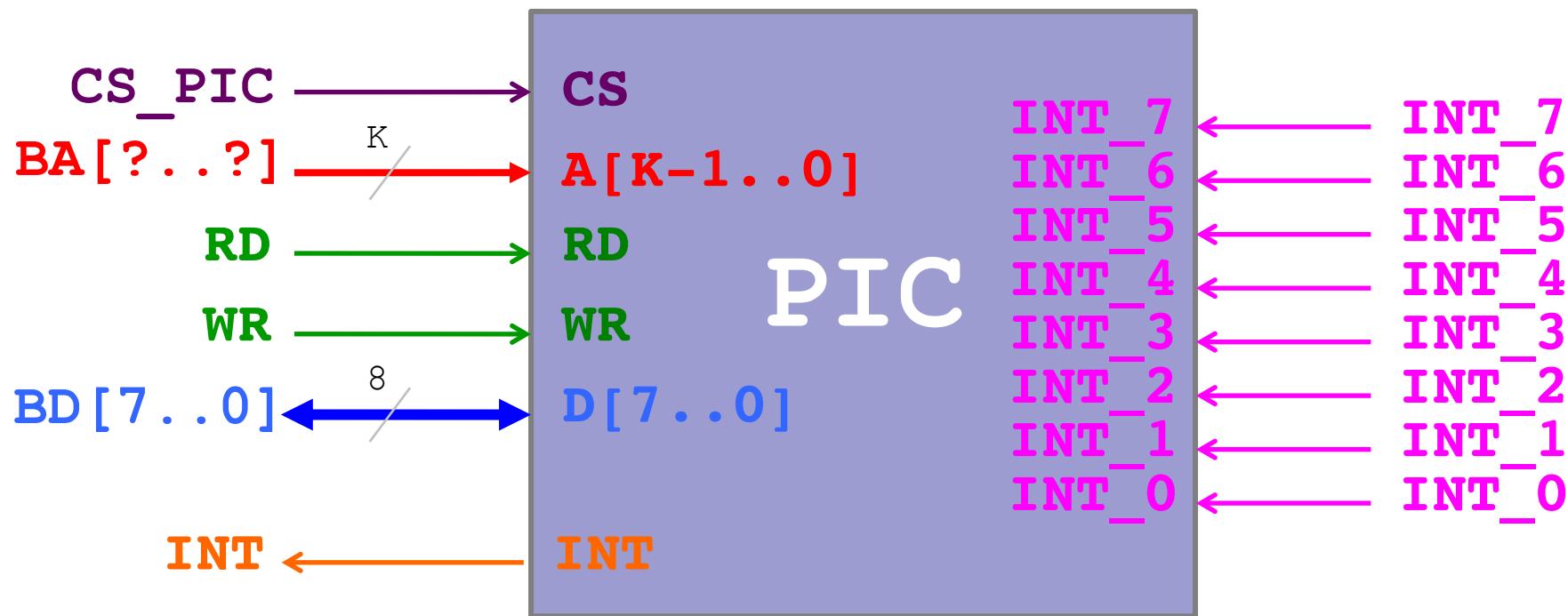
Nel caso di **multiple sorgenti di interruzione**, il codice di un tipico interrupt handler potrebbe avere la struttura seguente:

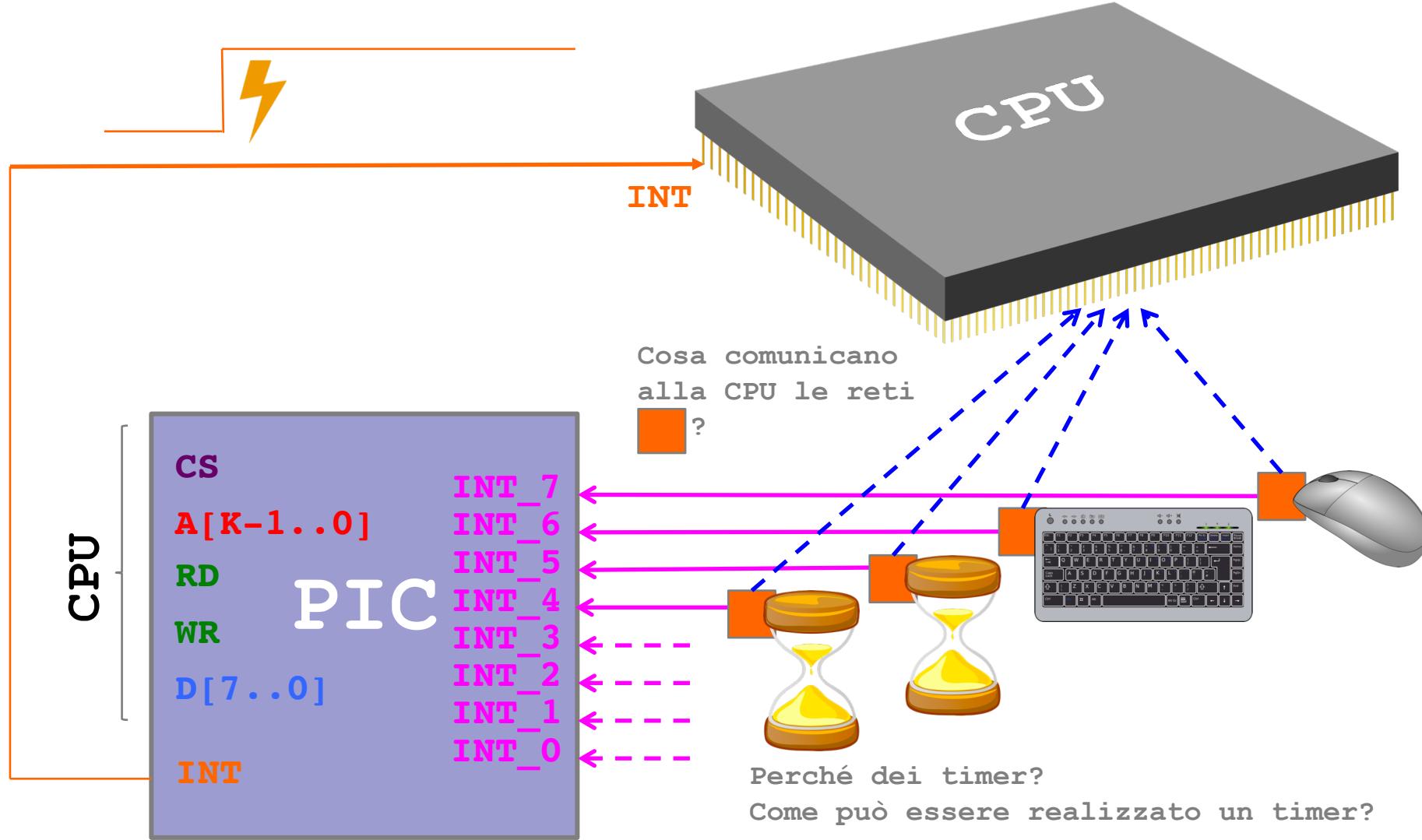
|            |   |   |
|------------|---|---|
| 00000000h  | <br><br> | <p>Preambolo</p> <p>; Istruzioni che salvano i registri<br/>; modificati dalle istruzioni seguenti</p> <p>; Identificazione dell'interrupt più<br/>; prioritario tra quelli asseriti</p> <p>; ripristina registri e salta al codice<br/>; dell' interrupt più prioritario</p> |
| XXXXXXXXXh |   | <p>RFE</p> <p>; salva registri modificati in seguito<br/>; codice handler_1</p> <p>; ripristina registri e ritorno (RFE)</p>  |
| YYYYYYYYYh |    | <p>RFE</p> <p>; salva registri modificati in seguito<br/>; codice handler_2</p> <p>; ripristina registri e ritorno (RFE)</p>  |

## Programmable Interrupt Controller (PIC)

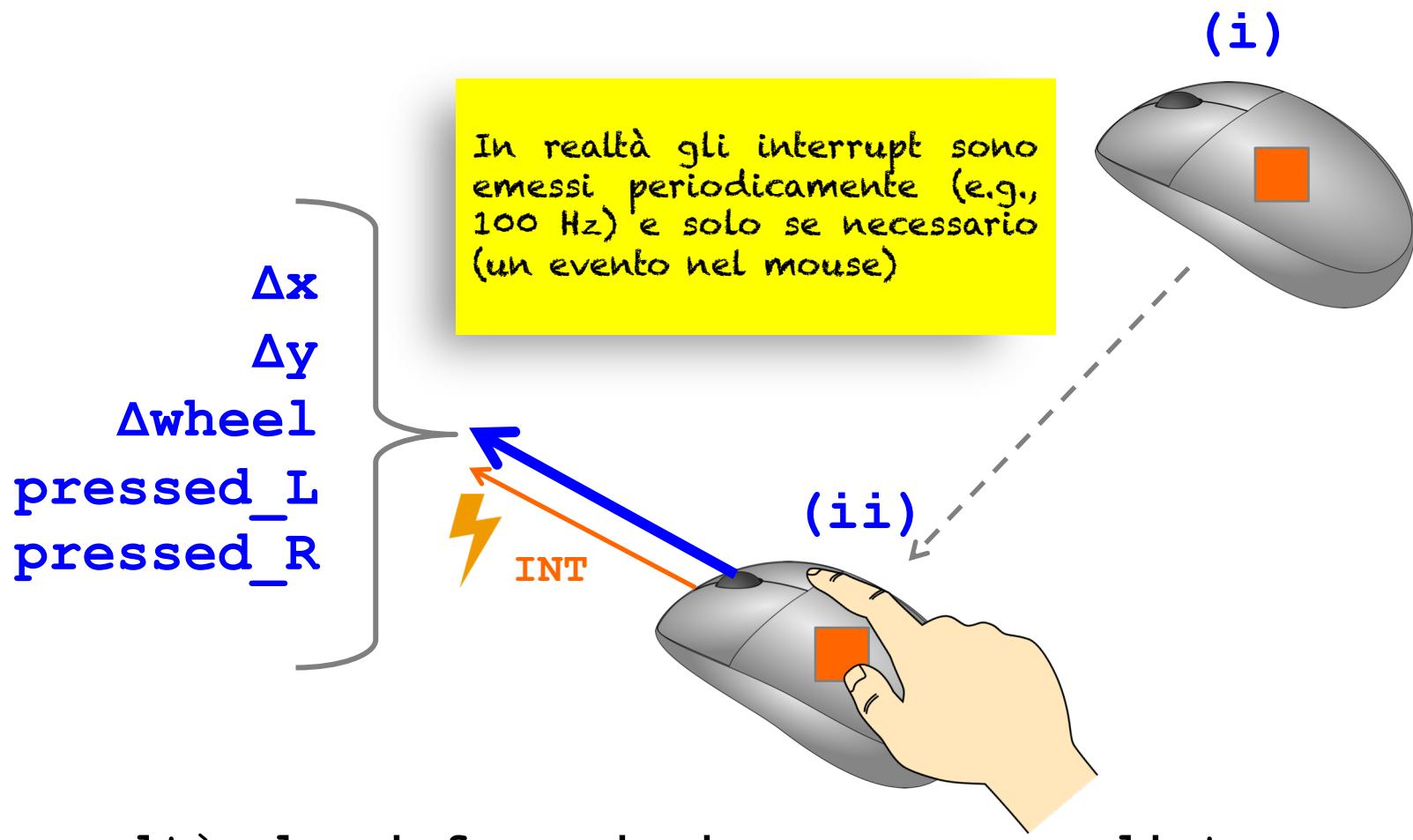
- Con la strategia mostrata nella pagina precedente è il software, interrogando ogni singola periferica, a dover determinare qual è l' interrupt più prioritario
- A tal fine sarà anche necessaria una opportuna infrastruttura hardware (i tri-state serviranno?)
- Tuttavia, è possibile velocizzare e semplificare le reti logiche di supporto a questo compito mediante l'utilizzo di un dispositivo ad hoc (PIC)
- Il PIC si occupa di gestire multiple sorgenti di interruzione e di fornire direttamente alla CPU (su richiesta) qual è il codice/indirizzo dell' interrupt più prioritario tra quelli asseriti in quel momento
- Tipicamente, in un PIC è possibile disabilitare le singole sorgenti di interruzione e stabilire il livello di priorità di ciascuna in accordo a varie politiche (priorità fissa, variabile, etc)

- La struttura di un ipotetico PIC potrebbe essere quella mostrata in seguito
- Le varie sorgenti di interruzione **INT[7..0]** sono inviate al PIC che si occupa di inviare la richiesta sull'unico pin **INT** del DLX
- Più avanti ne progetteremo uno molto semplice con funzionalità di base (abilita/disabilita **INT\_i**)





- Il PIC invia il segnale di **INT** e fornisce alla CPU, su richiesta, il codice dell'interrupt con priorità più elevata tra quelli asseriti in quel momento
- Perché nel PIC è presente anche il segnale **WR**?



- In realtà le informazioni sono convogliate su un canale seriale (USB, PS/2) per ridurre il numero di connessioni/fili
- Tuttavia, possiamo pensare per le nostre finalità che l'interfaccia mouse/CPU esponga i segnali di una porta di I/O standard (`CS`, `RD`, `WR`, `D[7..0]`, `indirizzi`)

# Device Manager

File Action View Help

VMWINDOWS81

- Direct memory access (DMA)
- Input/output (IO)
- Interrupt request (IRQ)
  - (ISA) 0x00000000 (00) High precision event timer
  - (ISA) 0x00000001 (01) Standard PS/2 Keyboard
  - (ISA) 0x00000006 (06) Standard floppy disk controller
  - (ISA) 0x00000008 (08) High precision event timer
  - (ISA) 0x00000009 (09) Renesas USB 3.0 eXtensible Host Controller - 0100 (Microsoft)
  - (ISA) 0x0000000C (12) Parallels Mouse Synchronization Device
  - (ISA) 0x0000000D (13) Numeric data processor
  - (ISA) 0x0000000E (14) ATA Channel 0
  - (ISA) 0x0000000F (15) ATA Channel 1
  - (ISA) 0x00000051 (81) Microsoft ACPI-Compliant System
  - (ISA) 0x00000052 (82) Microsoft ACPI-Compliant System
  - (ISA) 0x00000053 (83) Microsoft ACPI-Compliant System
  - (ISA) 0x00000054 (84) Microsoft ACPI-Compliant System
  - (ISA) 0x00000055 (85) Microsoft ACPI-Compliant System
  - (ISA) 0x00000056 (86) Microsoft ACPI-Compliant System
  - (ISA) 0x00000057 (87) Microsoft ACPI-Compliant System
  - (ISA) 0x00000058 (88) Microsoft ACPI-Compliant System
  - (ISA) 0x00000059 (89) Microsoft ACPI-Compliant System
  - (ISA) 0x0000005A (90) Microsoft ACPI-Compliant System
  - (ISA) 0x0000005B (91) Microsoft ACPI-Compliant System
  - (ISA) 0x0000005C (92) Microsoft ACPI-Compliant System
  - (ISA) 0x0000005D (93) Microsoft ACPI-Compliant System
  - (ISA) 0x0000005E (94) Microsoft ACPI-Compliant System
  - (ISA) 0x0000005F (95) Microsoft ACPI-Compliant System
  - (ISA) 0x00000060 (96) Microsoft ACPI-Compliant System
  - (ISA) 0x00000061 (97) Microsoft ACPI-Compliant System
  - (ISA) 0x00000062 (98) Microsoft ACPI-Compliant System
  - (ISA) 0x00000063 (99) Microsoft ACPI-Compliant System

## Interrupt non mascherabili (segnale NMI)

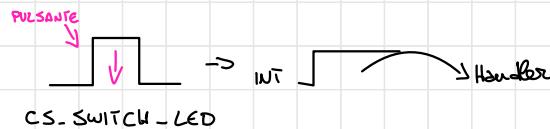
- In una CPU (ma non nel DLX) può essere presente un ulteriore segnale (in input) denominato **NMI** (*Not Maskable Interrupt*)
- A tale segnale sono collegate un numero limitato di sorgenti di interruzioni particolarmente critiche
- Per esempio, l'output di una rete che rileva e segnala una imminente perdita di alimentazione elettrica
- Una richiesta di interrupt inviata sul pin **NMI** non può essere ignorata (eventuali istruzioni che disabilitano gli interrupt non agiscono per questo segnale) e interrompe l'esecuzione di altri handler
- L'handler associato al pin **NMI** è a priorità massima e deve essere seguito nel minor tempo possibile

- Il segnale **NMI** va usato con cautela e solo per segnalazioni *critiche* alla CPU
- Nel caso del DLX utilizzeremo solo **INT**
- Se fosse disponibile, per la gestione del segnale **NMI** sarebbe necessario inserire le istruzioni nella prima parte del “preambolo” all’indirizzo **00000000h**, prima di gestire gli interrupt che sono inviati attraverso **INT**

## Esercizio

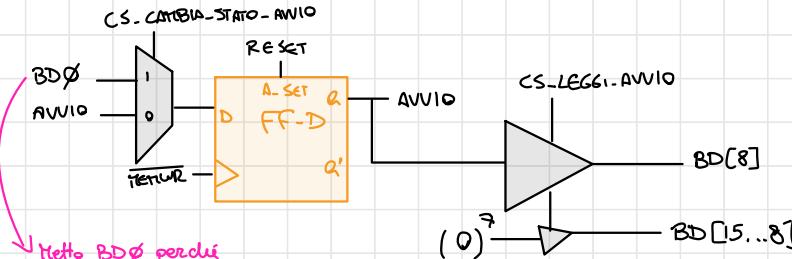
Progettare un sistema basato sul processore DLX, con un 1 GB di EPROM a indirizzi bassi e 512 MB di RAM a indirizzi alti. In tale sistema, utilizzando un pulsante, deve essere possibile accendere/spegnere un led mediante interrupt. All'avvio il led deve essere acceso.

Si faccia l'ipotesi che R29 e R30 possano essere usati senza la necessità di essere ripristinati.



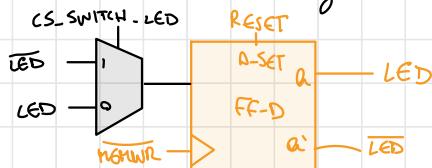
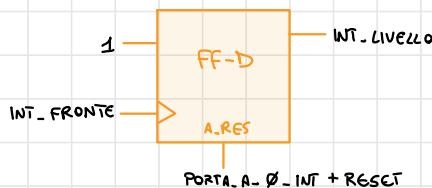
Ci serve capire se quando viene fatto il fetch all'indirizzo

O siamo nell'handler o all'avvio



Non dobbiamo mappare il pulsante perché genera un interrupt

Dobbiamo poter mandare a 0 l'interrupt e mandare a livello il segnale del pulsante



caso:

$$CS\_SWITCH\_LED = BA31 \overline{BA30} BE0$$

$$CS\_PORTA\_0\_INTERRUPT = BA31 \overline{BA30} \cdot BE3 \cdot MEMWR$$

$$CS\_CAMBIO\_STATO\_AVVIO = BA31 \overline{BA30} \cdot BE2$$

$$CS\_LEGGI\_AVVIO = BA31 \overline{BA30} BE1 \cdot MEMRD$$

$$CS\_RAM\_0 = BA31 BA30 BE0$$

È visualizzato sul bus dati [15..8]

$$CS\_RAM\_1 = " BE1$$

$$CS\_RAM\_2 = " BE2$$

$$CS\_RAM\_3 = " BE3$$

$$CS\_EPROM\_0 = \overline{BA31} BE0$$

$$CS\_EPROM\_1 = " BE1$$

$$CS\_EPROM\_2 = " BE2$$

$$CS\_EPROM\_3 = " BE3$$

- LVI

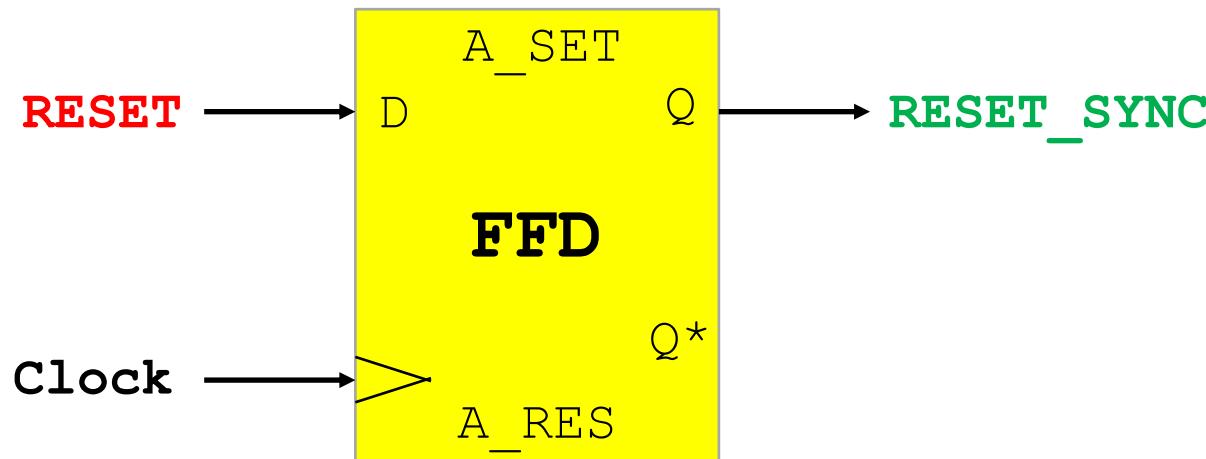
- Leggo se sono all'avvio

- Porto sono all'avvio a 0 nel caso in cui sono all'avvio

- Se sono in 0 e non sono all'avvio devo switchare il led e portare INT a 0

## Alcune considerazioni sul reset asincrono

L'applicazione di un segnale asincrono di reset, può portare a problemi di *metastabilità* nel momento in cui tale segnale viene posto al valore 0 (ie, quando si esce dal reset, assumendo che tale segnale sia attivo alto). Le problematiche sono analoghe a quelle evidenziate durante il campionamento di un segnale che non rispetta i tempi di *setup* e *hold*. Una possibile soluzione è la seguente:



Tuttavia, presenta dei problemi:

- E' sempre necessario un segnale di clock
- Quando **RESET** va a 1, **RESET\_SYNC** si asserisce (ie, diventa attivo) al primo fronte di clock

## ESERCIZIO INTERRUPT



CHIP SELECT →

0x4000 0000

0x4000 0001

0x4000 0002

0x4000 0003

CS\_RAM\_0 = BA31 BE0

CS\_RAM\_1 = " BE1

CS\_RAM\_2 = " BE2

CS\_RAM\_3 = " BE3

CS\_LED = BA31 BA30 BE0

CS.LEGGI\_AVVIO = " BE1 · MEXRD

CS.NO\_AVVIO = " BE2

CS.INT-∅ = " BE3 · MEXWR

CS.EPROM\_0 = BA31 BA30 BE0

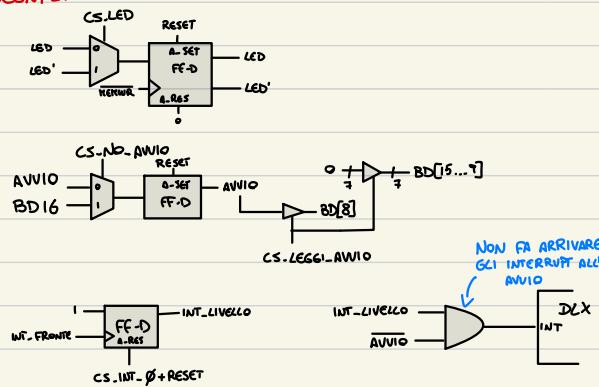
CS.EPROM\_1 = " BE1

CS.EPROM\_2 = " BE2

CS.EPROM\_3 = " BE3

VIA A 1 SOLO DURANTE  
QUEL CICLO DI BUS

SEGNALI



CODICE →

init: LH1 R29, 0x4000 ; inizializzo ∅

LBU R30, 0x0001(R29); leggo AVVIO

BEQZ R30, handler, se AVVIO = 0 salvo handler

SB R0, 0x0002(R29); metto AVVIO = 0

J main

handler: SB R30, 0x0000(R29); toggle Led

SB R30, 0x0003(R29); e metto INT e 0

RFE;

main: ...

Una soluzione che elimina i due problemi precedenti, e che garantisce un'uscita sincrona dal reset, è la seguente:

